On Efficiency and Effectiveness of Linear Function Detection Approaches for Memory Carving

Lorenz Liebler(✉) and Harald Baier

da/sec - Biometrics and Internet Security Research Group,
University of Applied Sciences, Darmstadt, Germany
{lorenz.liebler,harald.baier}@h-da.de

Abstract. In the field of unstructured memory analysis, the context-unaware detection of function boundaries leads to meaningful insights. For instance, in the field of binary analysis, those structures yield further inference, e.g., identifying binaries known to be bad. However, recent publications discuss different strategies for the problem of function boundary detection and consider it to be a difficult problem. One of the reasons is that the detection process depends on a quantity of parameters including the used architecture, programming language and compiler parameters. Initially a typical memory carving approach transfers the paradigm of signature-based detection techniques from the mass storage analysis to memory analysis. To automate and generalise the signature matching, signature-based recognition approaches have been extended by machine learning algorithms. Recently a review of function detection approaches claims that the results are possibly biased by large portions of shared code between the used samples. In this work we reassess the application of recently discussed machine learning based function detection approaches. We analyse current approaches in the context of memory carving with respect to both their efficiency and their effectiveness. We show the capabilities of function start identification by reducing the features to vectorised mnemonics. In all this leads to a significant reduction of runtime by keeping a high value of accuracy and a good value of recall.

Keywords: Memory forensics · Carving · Disassembly
Binary analysis

1 Introduction

The analysis of unknown binaries often starts with the examination of function boundaries. Functions are a fundamental structure of binaries and most often an initial starting point for advanced code analysis. As an important structural component of code, they give a schematic representation of the original high level semantics and provide a basis for further inferences. Whereas disassemblers are

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2019
Published by Springer Nature Switzerland AG 2019. All Rights Reserved
F. Breitinger and I. Baggili (Eds.): ICDF2C 2018, LNICST 259, pp. 3–22, 2019.
https://doi.org/10.1007/978-3-030-05487-8_1

capable of reliably decoding the instructions of a binary, the problem of function detection is still an ongoing field of research [1–3,13,14]. Binary analysis research claims that the problem is not yet fully solved. New techniques tend to improve in performance and generalizability, i.e., by the introduction of compiler- or even architecture-agnostic approaches [2,13]. Functions are used to infer the functionality of a given binary and thus, could be used to identify a unknown sample. In more general terms, two binaries that share many similar functions are likely to be similar as well [9]. Summarized, functions could be used to identify, distinguish or interpret unknown code sequences. Beside the field of extended binary analysis, those general tasks are obviously also relevant for other domains of application, where function detection techniques have to consider the present environmental circumstances and constraints.

The examination of process related code fragments is obviously one major benefit of memory based forensically investigations. After successfully reconstructing the running binary out of a process context, steps of binary analysis and reverse engineering are often followed [11]. However, the reconstruction could be hindered by malicious or legitimate changes. Additionally, remaining fragments of already terminated processes are possibly ignored, due to missing structural properties. In case of Linux operating systems the generation of an adequate memory profile could be cumbersome. The continuous development of operating system internals and its related structures require the constant maintenance of interpretive frameworks. Thus, even if the interpretation of operating system related structures is a fundamental component of memory analysis, carving could give a first solid impression or even be a last resort during examination.

In the course of function detection, machine-learning approaches have been proposed, which are trained to recognize signatures located in function prologues or epilogues [3,14]. Static function prologue signature databases have to be maintained over time and the detection performance of those techniques rapidly decreases for highly optimized binaries [4,6]. Machine learning based approaches try to generalize this task and automate the process of signature detection. Beside those signature related approaches, Andriesse et al. [2] introduced an compiler-agnostic approach in the context of extended binary analysis, which is mainly based on structural Control Flow Graphs. Moreover, their research showed significant worrying for all top-tier work on machine learning based approaches, mainly caused by the usage of a biased dataset. The compiler-agnostic approach has been extended by architecture-agnostic detection methods [13].

Considering the function identification process in the field of memory carving, the present conditions exclude most of the extended and agnostic approaches. Those have been proposed in the field of extended binary analysis and require steps of binary lifting, control flow analysis or value set analysis. In the course of memory carving, we further denote suitable function detection approaches as *linear* techniques. Those approaches do not rely on the reconstructability of binaries and could also be used for context-unaware memory analysis.

Contributions: We give an overview of recently discussed function detection techniques in Sect. 2 and categorize them into *linear* and *non-linear* applications. We outline the task of function detection and give insights in two particular machine learning techniques in Sect. 3. As recent publications underline the importance of non-biased data sets for the task of training and evaluation, we first summarize different data sets and the properties of our used set in Sect. 4. The analysis is later used for adapting our proposed models. In Sect. 5 we outline the concrete adaptations to the models and discuss the utilization of mnemonics only. This reduces the overall feature size and improves the runtime performance of the classification. We reassess the capabilities of *linear* function detection in Sect. 6 and emphasize our desired classification goals (i.e., a better performance in terms of recall and classification runtime). Our analysis underlines the applicability by reaching those goals even with significant reduced feature vectors. In contrary to recent publications, we consider runtime performance as an important constraint. We summarize our findings in Sect. 7.

2 Related Work

The enumeration of unknown functions was first established with the generation of signature databases. Signature databases focus on proprietary compilers, as open source compilers create an unhandable diversity of function prologues [4,6]. Especially in case of Linux operating systems, a database lookup of saved signatures during carving a memory image would be not feasible.

In Bao et al. [3] a weighted prefix tree structure was introduced to identify potentially function start addresses. Therefore, they "weight vertices in the prefix tree by computing the ratio of true positives to the sum of true and false positives for each sequence" in a reference data set. The authors additionally introduce an additional step of normalization, which improves precision and recall. The authors created a set of 2,200 Linux and Windows binaries. The executables were generated with different build settings, i.e., the authors used GNU `gcc`, Intel `icc` and Microsoft Visual Studio. In addition, multiple different optimization levels were selected during build time. Their approach, called `Byteweight`, was also integrated into the Binary Analysis Platform (BAP)[1].

In Shin et al. [14] the authors provide an approach for function detection based on artificial neural networks. The paper proposes a function detection technique with the help of Recurrent Neural Networks. In contrast to our work, the approach of [14] was performed without an additional step of disassembling or normalization. The authors point out that the tracking of function calls over large sequences of bytes is not feasible. In fact, recognizing entry and exit patterns by training with fixed-length subsequences is eligible. For training and testing the work is based on the same data set provided by Bao et al.

Andriesse et al. [2] claim that the work of Shin et al. and Bao et al. suffer from significant evaluation bias, as the most of the samples contain of large amounts of similar functions. The authors additionally mention that the viability

[1] https://github.com/BinaryAnalysisPlatform/bap (last access 2018-04).

of machine learning for function detection is not yet decided. The publication proposes a compiler-agnostic approach called `Nucleus`, which is mainly based on the examination of advanced control flow analysis and not relies on any signature information. The approach is not applicable in our context, as we want to inspect code fragments in large amounts of data within a sliding window: we have to consider the linear characteristic of our application, which in turn leads us to a signature-based or machine-learning based approach.

Potchik [13] introduced the integration of `Nucleus` in the Binary Ninja Reversing Platform[2] and proposes multiple strategies over multiple analysis passes than just rely on simple heuristics. The author mentions the possible reduction of complexity and scope reduction, by applying the technique with the highest confidence first. Similar to other fields, the approach proposes "a method to interpret the semantics of low-level CPU instructions" by the utilization of value-set analysis. The process of value-set analysis is performed on an extended intermediate language and thus, should be architecture agnostic.

3 Background

We first give a short introduction to the problem of function detection and describe the already introduced condition of *linearity* (Sect. 3.1). For a more detailed and formal explanation of the task of function detection, we refer to previous work [2,3,14]. The already introduced approaches in Sect. 2 are shortly discussed and categorized. We depict two applicable approaches for signature-based function detection and discuss their functionality: Recurrent Neural Networks (RNN) in Sect. 3.2 and Weighted Prefix Tress (WPT) in Sect. 3.3.

3.1 Linear Function Detection

The task of function detection is one of the main disassembly challenges [1]. The problem of function detection (e.g., with the help of static signatures) could be illustrated by the inspection of functions compiled with different optimization levels invoked to the compiler. The function structure and function prologue heavily changes due to optimization and compiler settings. An example could be seen in Fig. 1, which was adopted from [14]. The example shows the remarkable impact by simple adaptations of the compiler flags. Most of the instructions in the function prologues of `mul_inv` heavily differ from each other. The process of function detection itself is most often divided into subtasks. We stick to similar notation of previous work and refer to those publications [2,3,14].

The subtasks of function detection could be differentiated into *function start detection* and *function end detection*. The problem of *function boundary detection* is a superset of *function start* and *function end detection*. In the course of this work, we mainly focus on the task of function start detection. We borrowed most of the following definitions from Shin et al., where C defines a given code base of

[2] https://binary.ninja/ (last access 2018-04).

Listing 1.1: Original C Source

```
int mul_inv(int a, int b) {
  int b0 = b, t, q;
  int x0 = 0, x1 = 1;
  if (b == 1) return 1;
  while (a > 1) {
    q=a/b, t=b, b=a%b, a=t;
    t=x0, x0=x1-q*x0, x1=t;
  }
  if (x1 < 0) x1 += b0;
  return x1;
}
```

Listing 1.2: `gcc -O0`

```
<mul_inv>:
push %rbp
mov  %rsp,%rbp
mov  %edi,-0x24(%rbp)
mov  %esi,-0x28(%rbp)
mov  -0x28(%rbp),%eax
...
```

Listing 1.3: `gcc -O3`

```
<mul_inv>:
cmp   $0x1,%esi
mov   %edi,%eax
je    400878
cmp   $0x1,%edi
jle   400878
mov   %esi,%ecx
mov   $0x1,%r8d
xor   %edi,%edi
...
```

Fig. 1. Adopted example from [14] with a function written in C and compiled with two different levels of optimization (`gcc` 4.9.1).

a binary, which consists of several functions $f_1, \ldots, f_n \in F$. A function $f_x \in F$ consists of a sequence of instructions I, with $i_y \in I$. The task of detection could be simplified into three basic tasks for a given target function f_t:

1. **Function start:** The first instruction of $i_s \in I$ of $f_t \in F$ within C.
2. **Function end:** The last instruction of $i_e \in I$ of a $f_t \in F$ within C.
3. **Function boundaries:** The tuple of instructions, which define the boundaries of a function, i.e., determine $(i_s, i_e) \in I$ of $f_t \in F$ within C.

The scope of function detection normally sticks to disciplines of analysing stripped binaries in the context of reverse engineering, malware analysis or code reuse detection. However, most of these applications are not limited in their form of application and are not underlying any time constraints. As we transfer the function detection problem to the field of context-unaware domains, we have to consider the constraints of *linear* processing. A concrete field of application is the examination of function boundaries in the domain of unstructured and context-unaware memory analysis. As previous publications showed fundamental improvements in the case of compiler- and architecture-agnostic techniques, we emphasize the motivation behind signature-related techniques, as we could not rely on features like binary lifting, intermediate representations or control flow analysis. Our goal is the integration of a function start identification approach, which works on an instruction buffer within a single sliding window pass. So we refine the problem of function detection to be a *linear function detection* problem. This leads to the evaluation of signature-based approaches.

Shown in Table 1, different approaches have been suggested. In this work we focus on signature-related (Sig.) and linear approaches (Lin.). The approach of Shin et al. [14] is based on Bi-directional Networks, which are not applicable in our context. Those networks require the presence of a complete binary at application time. The lookup of signature databases on for Linux operating systems was already mentioned as impracticable. As we try to carve functions out of a memory, we could not rely on the recreation of a previously detected process.

Table 1. Overview of different approaches in the context of carving memory.

Approach	Sig.	Lin.	Comment
Guilfanov [6]	✓	✓	Impractical for diversity of Linux systems
Bao et al. [3]	✓	✓	Weighted Prefix Trees are applicable
Shin et al. [14]	✓	-	Bi-directional RNN not applicable
Andriesse et al. [2]	✗	✗	Process-context recreation needed
Potchik [13]	✗	✗	Process-context recreation needed

3.2 Recurrent Neural Networks

The authors of [14] provide a detailed overview of the different characteristics of Recurrent Neural Networks. Beside the work of [14], we refer to [12] and [8] for a detailed introduction. We outline differences between our approach and the approach of Shin et al. [14] later: The specific settings, hyperparameters and implementation details of our model are described in Sect. 5.

The processing of sequences with Recurrent Neural Networks is a promising strategy for many different fields. In contrast to feedforward neural networks, the cells keep different states of previously processed input and consider sequences which have an explicit or implicit temporal relation. A sample x_t of a sequence is additionally labeled with its corresponding time step t of appearance. Where a corresponding target y_t of labeled data also shares this temporal notation. The basic architecture of those nets could strongly vary for different fields of application. In Fig. 2 a simplified and unfolded model is shown, which takes multiple input vectors over time and outputs a single vector after several time steps. The term unfolding denotes the unfolding of the cyclic characteristic, by displaying each timestep. Each of the edges in between the columns span adjacent time steps. The model in Fig. 2 depicts a many to one relation.

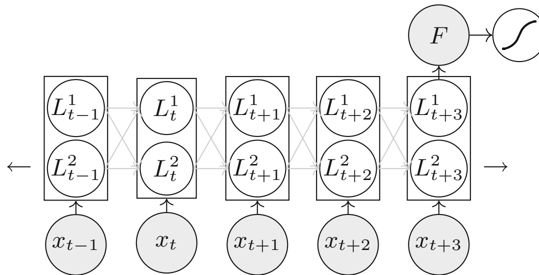

Fig. 2. Simplified RNN model with two LSTM-layers, where the model represents a many to one structure. The final state is processed by a Fully Connected Layer F and a sigmoid layer, which outputs a probability.

Long Short-Term Memory (LSTM): Training Recurrent Neural Networks bares several pitfalls. Namely, the problems of vanishing and exploding gradients. There are different extensions of RNNs which try to consider those issues, one of those specimens are LSTM based Networks [5,8]. Those networks replace traditional nodes in the hidden layers with memory cells (see Fig. 3). Each cell consists of an input node (g) and internal state (c). The memory cells contain self-connected recurrent edges, to avoid vanishing or exploding gradients across many time steps. Those edges are named gates, where each of the LSTM cells has a forget gate (f), an input gate (i) and an output gate (o). A multiplication node ($\prod$) is used to connect those components with each other. The final memory cell reaches an intermediate state between long-term and short-term memory of classic RNNs. Those types of cells outperform simple RNNs in case of long-range dependencies [12]. The original work of Shin et al. is based on Bi-directional RNNs. The authors mention that those models are applicable in the context of having access to the entire binary at once. As we discuss the applicability of RNNs in the context of linear processing large amounts raw data, we have to consider the temporal component and focus on a LSTM based model.

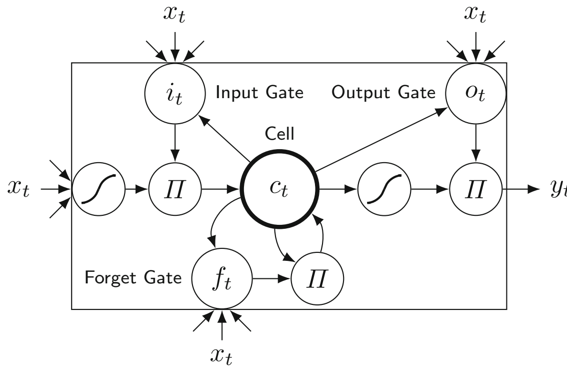

Fig. 3. LSTM Memory cell proposed by [5], which extends the model of [8] with additional forget gates (f_t).

Training and Classification: There are different strategies for training and updating a neural model. The predominant algorithm for training is backpropagation, which is based on the chain rule. The algorithm updates the weights of the model by calculating the derivative of the loss function for each parameter in the network and adjusts the values by the determined gradient descent. However, the problem is a NP-hard Problem and different heuristics try to avoid stucking in a local minimum. A common strategy for training is stochastic gradient descent (SGD) using mini-batches. In the course of RNNs with time-related connections between each time step, the process of training is often denoted as backpropagation through time (BPTT) [12]. Bao et al. [3] proposed a flipped order of prologues during processing to support the training phase. Additionally, the

work shows a better performance for bidirectional structures than unidirectional structures like LSTM. Each model was trained with 100.000 randomly-extracted 1000-byte chunks. A one-hot encoding converts a byte into a $\mathbb{R}^{256}$ vector.

3.3 Weighted Prefix Tress

Bao et al. [3] introduced the application of weighted prefix trees for the task of binary function detection, called `Byteweight`. The approach uses a weighted prefix tree, and matches binary fragments with the signatures previously learned. The path from the root node to the leaf node represents a byte sequence of instructions. Inside the tree, the weights are adapted to the previously processed ground truth. In the original implementation an additional step of value-set analysis and control flow recovery process is proposed for boundary detection.

Byteweight: Similar to this work and the work of Shin et al. [14], Byteweight focuses on the task of function start identification. More formally, the authors denote the problem as simple classification problem, where the goal is to label each byte of a binary as either function start or not. Their approach was demonstrated on raw byte sequences and previously disassembled sequences. The reference corpus is compiled with labelled function start addresses. In contrary to raw bytes, the usage of normalized disassembled instructions showed a better performance in case of precision and recall. The authors proposed a twofold normalization: immediate number normalization and `call-jump` instruction normalization. In Fig. 4 an overview of the normalized prefix tree is given. A given sequence of bytes or instructions is classified by inspecting the corresponding terminal node in the tree. As soon as the stored value exceeds a previous defined threshold t, the sequence is considered as a function start. We do not consider subsequent steps of advanced control flow graph recovery as proposed by the authors.

Training and Classification: A corpus of input binaries is used during the learning phase. The maximum sequence length l defines the upper bound of the resulting trie height. The first l elements are used for training, where elements

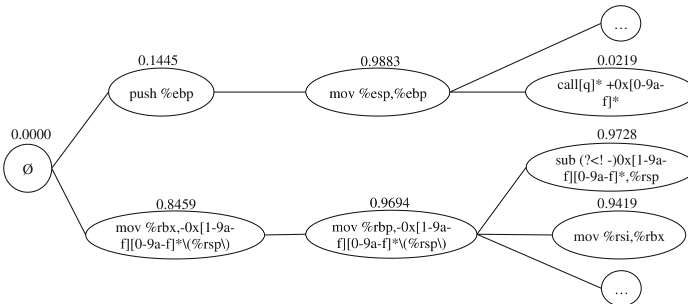

Fig. 4. Example of normalized prefix tree proposed by Bao et al. [3].

could be disassembled instructions or the raw bytes itself. The likelihood that a sequence of elements corresponds to a function start (i.e., represented as a specific path in the trie) is saved in each corresponding node as specific weight. Considering the example in Fig. 4, the instruction `push %ebp` were truly function starts in 14.45% of all cases. The weights of a prefix are lowered if they do not correspond to a function start. As described in Eq. 1, the weight of a specific node W_n is the ratio between positive function starts (T_+) and all matches ($T_+ + T_-$). The classification of an input sequence is performed by matching the given elements against the tree. The weight of the last matching terminal node, describes the final weight of a sequence and is compared to t. For the process of training and classification, the authors proposed an input size of $l = 10$ consecutive instructions and a threshold of $t = 0.5$.

$$W_n = \frac{T_+}{T_+ + T_-}. \tag{1}$$

4 Training and Evaluation Data Sets

In recent publications, different sources of ground truth binaries have been proposed and criticized. In this paragraph we give a short overview of the different sources and outline some details of capacity and source. As we focus on the domain of Linux executable binaries, we formally introduce ELF binaries contained in different test suites. A comprehensive overview of the different test suites is given in Table 2 which have been public available[3,4,5] at the time of writing.

Table 2. Overview of different evaluation datasets [13].

Source	System			Description (ELF, Linux)
	WIN	LIN	OSX	
`Byteweight`	✓	✓	✗	ELFs (129): `coreutils`, `binutils` and `findutils`; used by Bao et al. [3] and Shin et al. [14]
`Nucleus`	✓	✓	✗	ELFs (521): real-world applications and the SPEC CPU2006 Benchmark Suite; see Table 3 for details
`CGC Corpus`	✓	✓	✓	ELF binaries of custom-made programs specifically designed to contain vulnerabilities

[3] https://github.com/Vector35/function_detection_test_suite (last access 2018-04).
[4] https://github.com/trailofbits/cb-multios (last access 2018-04).
[5] http://security.ece.cmu.edu/byteweight/ (last access 2018-04).

As already outlined in the introduction, the work of Bao et al. [3] and Shin et al. [14] are criticized by Andriesse et al. [2] for using a biased data set, with a large amount of overlapping and similar functions. Andriesse et al. [1] outlined that the average binary in their SPEC-based test suite contains less than 1% of shared functions, not considering bootstrap functions. We base our analysis on the data set introduced by Andriesse et al. [1] and perform a detailed examination of the function structures in the following paragraphs.

The `Nucleus` data set consists of approximately 4.2 GiB precompiled ELF files and its corresponding ground truth assembly structure. The process of data set generation depends on some major parameters: operating system, instruction set architecture, language, compiler and optimization level. The 521 binaries consist of the SPEC CPU2006 Benchmark Suite and some real-world application written in C and C++. The samples are compiled for x86 and x64 with five different optimization levels (O0-O3 and Os). The set contains dynamically and statically linked binaries, where some of them are stripped and some are equipped with symbols. For further details on the construction of the ground truth we refer to [1]. An overview of the binaries is given in Table 3. In the following paragraphs we give a detailed introduction of the data set. Therefore, we focus on three properties of our used data set: *Function Sizes*, *Function Prologue Distribution* and *Mnemonic Distribution*. As already described in the introduction, the examination of the underlying code structure should give us additionally insights for better design and parameter decisions.

Table 3. `Nucleus` - ELF ground truth obtained by [1] (gcc-510, llvm-370).

Samples	Arch		Compiler		Language		Optimization				
	32	64	gcc	llvm	C	C++	O0	O1	O2	O3	OS
SPEC	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
glibc	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
Server	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗
Count	200	321	321	200	360	140	100	100	100	100	100

Function Sizes: The examination of the frequencies of different function sizes is required for further analysis and inferences. The function size defines the possible size of extracted features, before a single feature vector overlaps with a subsequent function start. In addition, the function size gives better insights into the possible dimension of further function prologues examinations. We examined the function size (in bytes) of the different binaries. As could be seen in Fig. 5, the function sizes vary for different compiler and optimization levels. The median function size illustrates that in every language, compiler and optimization setting, the amount of small functions (i.e., smaller than 200 Bytes) is significant. The average value of function sizes outlines the presence of large functions, where in all settings the average size is always lower than 800 Bytes.

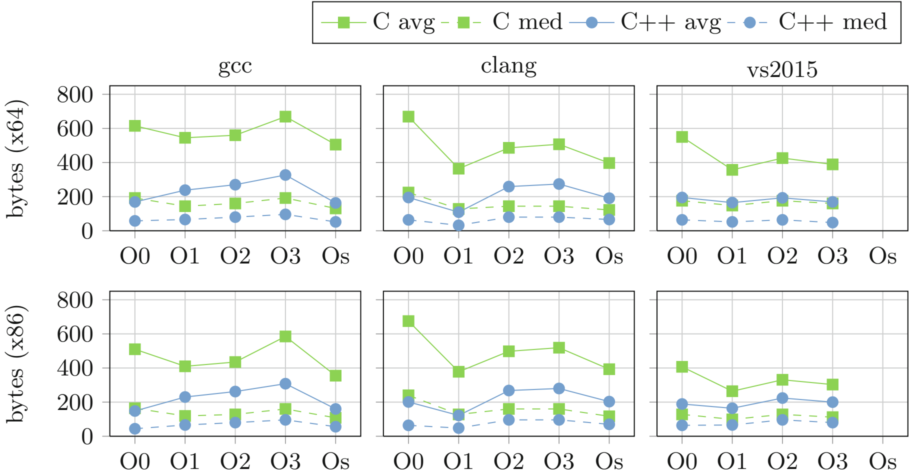

Fig. 5. Average and median function size in bytes.

Function Prologue Distribution: To gain a better understanding of the function detection problem with the help of signature-based detection mechanisms, we examined the present ground truth set and the distribution of common function prologues. We extracted the functions of each binary and aggregated them into a comprehensive set. Figure 6 illustrates the population of function prologues by comparing the ratio η between distinct function prologues ρ_d to the overall amount of function prologues ρ for a specific language, compiler and optimization level (for further details see Eq. 2).

$$\eta = \frac{\#\,\text{distinct function prologues}}{\#\,\text{function prologues}} = \frac{\rho_d}{\rho} \tag{2}$$

We discriminate the function prologues by its consisting number of instructions i, which are considered and which have been decoded to a single mnemonic. The plot underlines the common axiom that function prologues strongly vary for different compilers, languages and optimization levels. The plot visualizes the impact on the diversity of the prologue instructions in dependency to the selected optimization levels and helps to argue about an appropriate input size.

With the x86 instruction set an instruction could have variable-length, where one instruction could vary between one and fifteen bytes. For further details we refer to the Intel Instruction manual[6]. Considering the previous examination of function sizes and the median value of 200 Bytes per function, we do not expect all instructions of a prologue sequence to reach the maximum amount of fifteen bytes. It is clear that a large chosen amount of input mnemonics raises the diversity the model has to deal with, but also increases the possible classification quality. Considering the plots in Fig. 6, the diversity significantly increases

[6] https://software.intel.com/en-us/articles/intel-sdm (last access 2018-04).

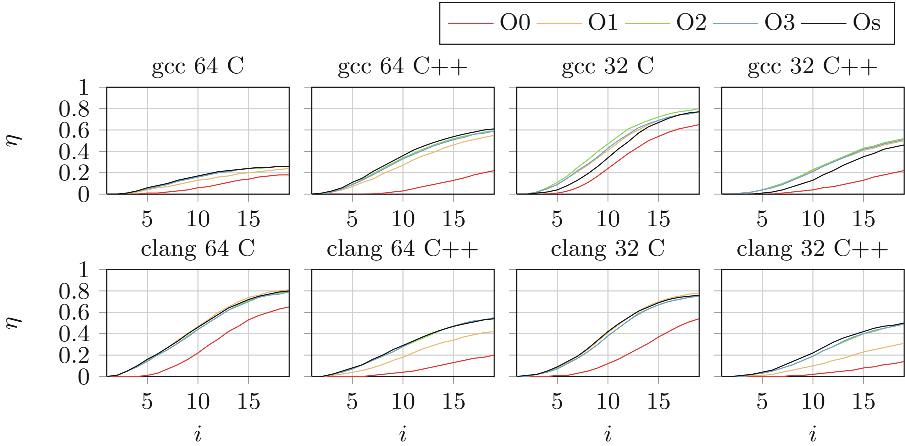

Fig. 6. Illustration of distinct function prologues; η denotes fraction of distinct function prologues to the number of all functions for i instructions (mnemonic).

for several settings, even after a considerable short amount of consecutive prologue instructions. For example, inspecting `clang-32-C` without optimization (i.e., `O0`), approximately 40% of all function prologues of 10 consecutive instructions represent a distinct instruction sequence. Figure 6 also underlines that non-optimized binaries (`O0`) often share the same beginning instructions (mnemonic).

Mnemonic Distribution: We use the ground truth of assembly files to determine the distribution of mnemonics in the used set. Additionally, we extract the bigrams of mnemonics, which could be often found in the course of assembly based code and similarity analysis. The following values give us an initial overview of the mnemonics distribution. For details see Table 4. Roughly spoken it is an overview of the instruction distribution of already decoded byte sequences. We splitted the set of assemblies by its architecture and determined the *total* amount of unigrams and bigrams. In our case a unigram consists of a single mnemonic. The *total* amount of occurring mnemonics also represents the total amount of occurring instructions. The column of *distinct* values describes the set of all occurring mnemonics. The columns *max*, *mean* and *min* describe the assignment of the *total* amount of instructions to each *distinct* occuring unigram or bigram. In detail, *the most frequently occurred mnemonic* in the set of

Table 4. Overview of unigram and bigram mnemonic counts.

	32 bit (200 files)				64 bit (321 files)			
	Total	Distinct	Max	Mean	Total	Distinct	Max	Mean
Unigrams	35,232 k	322	11,714 k	1531	61,441 k	436	21,627 k	1859
Bigrams	35,232 k	11632	5,889 k	17	61,441 k	16059	10,360 k	28

32 bit files, namely mov, represents 11,714,270 instructions. Thus, mov represents approximately 33.25% of all instructions in the course of 32 bit files.

5 Linear Function Detection

In this Section we discuss concrete adaptations and realizations of linear function detection techniques. Beside the reassessing of machine learning-based approaches we aim for a reduction of the used feature sizes to improve the theoretically runtime performance. Therefore, we introduce an additional step of approximate disassembling by the usage of an approximate disassembler [10]. In previous work the created pipelines are partially based on the processing of features on a byte-level. The input sequences for training and evaluation are one-hot encoded into $\mathbb{R}^{256}$. As our approach is based on integerized mnemonics, the distinct occurring mnemonics define our underlying vocabulary size.

We first describe the general pipeline and the used set for training and evaluation (Sect. 5.1). We explicitly address the problem of imbalanced classes, i.e., the ratio between function starts and general offsets. Afterwards, we introduce the proposed models and concrete adaptations (Sects. 5.2 and 5.3).

5.1 General Pipeline of Feature Extraction

In Fig. 7 an overview of the single steps of feature extraction is given. We only considered allocable code sections (i.e., .text) of the used ground truth ELFs [1]. Thus, we filtered all offsets which are known to be data ❶.

Similar to Bao et al. [3] and contrary to Shin et al. [14] we propose a layer of disassembling for further feature extraction ❷. We additionally reduce the used vectors to mnemonics only. We decode the raw byte sequences into an approximate disassembly with an integerized mnemonic for each instruction. Beside the reduction of variances in the underlying byte structure, this additionally reduces the overall amount of data which needs to be processed and saved. The classification is not performed at each byte offset, but rather at every instruction offset. Thus, we reduce the overall vector input size and therefore the runtime performance.

As we have to deal with an heavily imbalanced set of classes, we first determine the positive (function beginnings) and negative classes (code offsets) for each file ❸. For each class, we created vectors of N_{max} consecutive instructions at each instruction offset, represented by a single and integerized mnemonic. Considering the function prologue distribution in Sect. 4, most of the displayed distributions showed tendencies to stabilize after the first 20 instructions. The future vector size should also consider the determined average and median function sizes, as we try to avoid feature vectors which overlap into subsequent functions. Where Shin et al. proposed the processing of 1000-byte chunks for RNN-based classification and Bao et al. suggested the usage of 10 consecutive instructions for the creation of Weighted-Prefix-Trees, we considered a maximum

vector size of $N_{max} = 20$ mnemonics for creating our data set. This empowers us to vary different input sizes during different model evaluation passes.

We created two sets for both classes and performed an additional step of deduplication for each class over the whole set. This results in two sets of distinct feature vectors ❹. In Table 5 an overview of the imbalanced distribution of classes is given. We also examined possible overlaps between each class, i.e., vectors of mnemonics which occur in the positive T_+ and negative class T_-. This is obviously caused by the strong reduction to mnemonic feature vectors. We relabel the negative classes of $T_+ \cap T_-$ to provoke false positives. We outline this decision in Sect. 6. The two distinct features have been shuffled before and splitted into a training and test set ❺. In the case of RNN based training, we additionally added an step of oversampling T_+ and undersampling T_- ❻.

Table 5. Overview of class distribution, i.e., number of distinct function starts (T_+) and distinct inner function offsets (T_-) for all used 64 bit binaries.

Set	T_-	T_+	$T_+ \cup T_-$	$T_+ \cap T_-$
Count	18,575,407	207,714	18,779,238	3,883

5.2 RNN Model Adaptation

In the following subsection, we describe the final model settings for our linear RNN-based classification. The basic steps of processing are displayed in Fig. 8. The extracted and integerized mnemonics are transformed into a one-hot encoding and fed into the RNN for further processing. This pipeline is consistent for all further training and evaluation steps performed on our RNN settings.

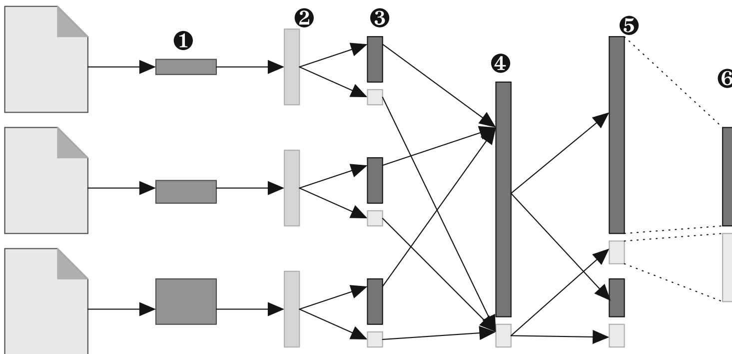

Fig. 7. Simplified overview of the general pipeline of feature extraction.

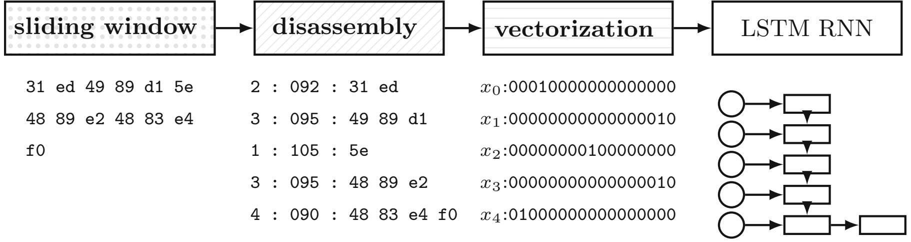

Fig. 8. The general processing pipeline during RNN-based classification.

Input Vectors: We are not expecting to train with an input time series larger than 20. In particular, we try to avoid long series for classification, to minimize the impact on runtime performance later. After the examination of distinct function prologues shown in Fig. 6, we varied the input vector size between 16 and 20 consecutive and integerized instructions. For the final model, we depicted the vector size of instructions as $N_i = 16$. Similar to Shin et al. we reversed the order of our input vectors, which showed a significant improvement in nearly all of our evaluations. Summarized, the network expects a reversed one-hot encoded input vector $x_t \in \mathbb{R}^K$ for each decoded instruction i_t, where K describes the current size of the vocabulary.

Hidden Cells and Layers: As already described in Sect. 3, we use Long Short-Term Memory cells for creating our RNN model. In detail, we use a two-layered model with a different number of hidden cells. To reduce the complexity we use LSTM in contrary to bi-directional RNNs as suggested by Shin et al.. We initially compared two-layered RNNs with one-layered models. In most of the cases a two-layered model improved the accuracy, which made us to perform all of the further proceedings with a two-layered model. We vary the number of hidden cells in the subsequent evaluation between 32 and 512 for our two-layered setting. We finally choose the amount of $N_h = 256$ hidden cells.

Training and Optimization: Similar to Shin et al. [14] we performed our steps of optimization by the usage of *stochastic gradient descent*. We tried different concepts of gradient adaptation and initially used *RMSProp* for optimization with different initial learning rates. In the further proceedings of the evaluation, we switched to *AdamOptimizer* which showed a similar performance. The gradients were updated with the help of mini-batches, where the size of the batches was also varied during evaluation and performance tuning. After several manual test runs, we set the final batch size to $N_b = 2048$.

Dropout: We reduced the risk of overfitting our models by adding an intermediate Dropout layer. A Dropout step randomly turns off activations of neurons between our two LSTM layers. The Dropout is not applied on the recurrent connections itself [7]. A defined value of $N_d = 0.75$. sets the probability if a connection is *not* deactivated. We keep this value for all of our trained models.

Output Layer: For a binary classification of the function starts we process the output of the final LSTM state by a fully connected layer. The final layer of the setup is a sigmoid layer, which is used for transforming our output into a binary classification. A single probability is generated with the help of an additional *sigmoid cross entropy* layer or *weighted cross entropy* layer. With the usage of a *weighted cross entropy* layer, we could additionally set the focus on improving our final recall or precision rate.

Training and Sampling: To handle the heavily imbalanced ground truth, we performed an additional step of oversampling the positive and undersampling the negative class. Those steps had an remarkable influence on the performance in case of precision and recall. During training we randomly selected the half of the negative class member (inner function offsets) and oversampled the positive class members (function starts) to an even amount.

5.3 Weighted Prefix Tree Adaptation

The major adaptation in case of our proposed Weighted-Prefix-Tree model, is the utilization of single mnemonic representatives instead of raw bytes or normalized disassembly instructions. The basic steps of processing are displayed in Fig. 9. We selected different lengths for trie creation up to $l_{max} = 20$ instructions. The mentioned vocabulary size K (see Sect. 4) gives us the theoretical upper bound for the possible trie size. The implementation of evaluation was realized with Python, not yet considering runtime in first case. We depict an initial threshold value of $t = 0.5$ and $l = 10$, which was proposed by Bao et al. [3].

Tree Pruning: Similar to the original approach we additionally performed a step of tree pruning. In detail, we deleted all intermediate nodes with no negative counts ($T_{-} = 0$) and thus, all first occurring intermediate nodes with $W_i = 1$ are transformed to terminal nodes. This reduces the overall tree size from approximately 1.9 million nodes to 264,834 nodes.

Fig. 9. The general processing pipeline during WPT-based classification.

6 Evaluation

In this section, we discuss the different evaluations of our proposed models. We inspect the time of creation and training our models, without considering the pass of feature extraction (Sect. 6.1). Afterwards, we examine the performance in terms of accuracy, precision and recall for both models (Sect. 6.2). Finally, we

discuss in detail the possible application of weighted prefix trees (Sect. 6.3). Our evaluation aims to answer the following questions:

1. Could we use only mnemonics for the task of function start identification?
2. Which of the considered linear techniques is better suited for our context?

Classification Goals: We consider our models for the fast identification of function starts in large portions of raw data. As described in Sect. 5.1, the reduction of our proposed features leads to overlapping class members. We relabelled those members to positive members and further treat recall more important than precision: we accept higher values of false positives by lower values false negatives with a constant high value of true negatives. We propose an additional step of classification for lowering the value of false positives afterwards.

Methodology: To argue if our models generalise the task of function detection, we performed a 10 fold validation by dividing our set of *distinct* integerized mnemonic vectors into ten equally sized sub-sets. We used nine of those subsets for training and one for evaluation. In the case of WPT-based training, we repeated this task 10 times. In our current evaluation we focus on 64 bit ELF binaries of the Linux operating system. Our used test set consists of nearly 1.9 million unique vectors with approximately 20 k positive cases. The set of training consisted of 16 million negative and nearly 190k positive class members.

6.1 Training Performance

First, we examine the general runtime performance in case of model creation and training. Both models have been fed with the already prepared integerized mnemonic vectors and a maximum length of 20. In the case of our RNN model, we already cropped the initial feature vectors to a maximum size of 16.

As already described in Fig. 8, in case of training our proposed RNN the processing began with the reversing of the input vector and the one-hot encoding. Considering those steps, the training of our model took approximately 2.5 hours for one epoch. As we trained our model for 10 to 20 epochs, the process most often took *several days* on a machine with 64 cores and more than 256 GiB of RAM. We utilized Tensorflow and implemented our model in Python.

In contrary, the creation of the weighted prefix tree was performed on an ordinary Laptop with Intel Core i5 2x 2,2 GHz Processor and 8 GiB RAM. The time of building the initial tree, calculating the final weights, pruning the tree and performing an additional step of lookup took approximately *180 s* in total. Our current prototype implementation was realized in Python.

Table 6. Performance of the function start identification for x86-64 ELF binaries. (*:average values of 10-fold cross validation)

Model	Accuracy	Precision	Recall	Population	TP	TN	FP	FN
WPT*	0.9943	0.7532	0.7233	1,878,311	15,023	1,852,616	4,924	5,747
RNN	0.9861	0.4300	0.7674	1,878,016	15,940	1,836,117	21,128	4,831

6.2 Function Start Identification

For our final models, we used the in Sects. 5.2 and 5.3 described settings. As could be seen in Table 6 the application of our RNN performs similar to our proposed WPT. In both cases we gain a high value of accuracy with lower values of precision and recall. However, the results of the WPT clearly outperforms the RNN in case of precision. As already described in Sect. 5.2, the adaptation of the underlying RNN model could influence the classification in terms of focusing on better values of recall or precision. However, in contrary to our WPT implementation, the adaptation of those RNN parameters must be done before the training of the model. The WPT approach empowers to influence the value of recall by parametrizing the lookup, which is more flexible than approaching the perfect parameters by multiple time consuming training passes.

6.3 Examination of WPT

As the previous evaluations underlined, we achieved similar or better classification capabilities in less training time by the utilization of our WPT model. For the WPT model, we could additionally adapt the classification goal during a lookup pass. This leads to a detailed examination of our WPT model.

To improve the quality of WPT-based classification, we examined the different parameters and their influence on the overall classification task. In detail, we varied different parameters, like the maximum sequence length l and the threshold t. In addition, we introduce a third parameter m, which denotes the minimum required length of a matching sequence inside the tree. After the inspection of the Prologue Distributions in Sect. 4, we selected a range from 0 to 18 (instructions) for the parameters m and l. Respecting our classification goal, we focus on runtime performance by accepting a higher amount of false positives instead of false negatives. Inspecting the classification in Table 7, we could argue that the reduction of false negatives could be achieved by increasing the considered vector size l. The processing time for the nearly 1.9 million test vectors was approximately 12 s for all of the parameter variations. We should mention that during

Table 7. WPT classification by varying different parameters t, l and m.

		Accuracy			Precision			Recall			Time (sec.)		
t	$m \backslash l$	10	14	18	10	14	18	10	14	18	10	14	18
0.6	0	0.9944	0.9944	0.9944	0.7867	0.7666	0.7589	0.6829	0.7222	0.7278	11.043	11.134	11.152
	4	0.9944	0.9944	0.9944	0.7889	0.7685	0.7608	0.6793	0.7186	0.7243	11.063	11.163	11.154
	8	0.9933	0.9933	0.9933	0.8203	0.7893	0.7785	0.5067	0.5460	0.5517	11.066	11.135	11.371
0.5	0	0.9937	0.9936	0.9934	0.7098	0.6868	0.6786	0.7385	0.7743	0.7800	11.408	11.853	11.871
	4	0.9937	0.9936	0.9935	0.7127	0.6893	0.6810	0.7359	0.7718	0.7775	11.429	11.084	11.564
	8	0.9930	0.9928	0.9927	0.7504	0.7147	0.7029	0.5542	0.5901	0.5957	11.587	11.556	11.422
0.4	0	0.9932	0.9929	0.9928	0.6686	0.6476	0.6415	0.7691	0.7974	0.8024	11.316	11.481	11.881
	4	0.9933	0.9930	0.9929	0.6752	0.6535	0.6472	0.7648	0.7931	0.7981	11.804	11.653	11.326
	8	0.9929	0.9926	0.9925	0.7283	0.6931	0.6835	0.5749	0.6032	0.6082	11.326	11.908	11.950
0.3	0	0.9917	0.9913	0.9912	0.5942	0.5757	0.5707	0.8046	0.8314	0.8361	11.297	11.418	11.539
	4	0.9920	0.9916	0.9915	0.6072	0.5875	0.5821	0.8001	0.8268	0.8315	11.505	11.430	11.414
	8	0.9924	0.9920	0.9919	0.6758	0.6414	0.6327	0.6064	0.6332	0.6379	11.055	11.135	11.277

our initial implementation we did not focus on any runtime optimizations, which could be further improved.

7 Conclusion

In this paper, we inspected the capabilities of linear function detection and underlined the need of signature-based detection methods in the course of memory carving. After performing a detailed analysis of our underlying ground truth, we introduced several considerations and model adaptations. The main adaptation of our approach is the utilization of mnemonics only.

Our analysed models showed good classification results in term of accuracy, where we achieved for RNN and WPT based models a value of accuracy above 98%. The utilization of menmonic-based weighted prefix trees showed good capabilities for our considered context of application. The application of WPT performs the classification of 1.9 million offsets in 12 s and reaches with simple parameter adaptations an acceptable value of recall beyond 80%.

Considering the mentioned techniques within multiple steps of classification, our proposed WPT could be used for the fast identification of possible function starts and drastically reduces the amount of offsets which need to be reclassified.

Acknowledgement. This work was supported by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP (www.crisp-da.de).

References

1. Andriesse, D., Chen, X., van der Veen, V., Slowinska, A., Bos, H.: An in-depth analysis of disassembly on full-scale x86/x64 binaries. In: USENIX Security Symposium (2016)
2. Andriesse, D., Slowinska, A., Bos, H.: Compiler-agnostic function detection in binaries. In: IEEE European Symposium on Security and Privacy (2017)
3. Bao, T., Burket, J., Woo, M., Turner, R., Brumley, D.: Byteweight: learning to recognize functions in binary code. In: USENIX (2014)
4. Eagle, C.: The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler. No Starch Press, San Francisco (2008). ISBN 1593271786, 9781593271787
5. Gers, F.A., Schmidhuber, J., Cummins, F.: Learning to Forget: Continual Prediction with LSTM (1999)
6. Guilfanov, I.: IDA Fast Library Identification and Recognition Technology (Flirt Technology): In-depth (2012)
7. Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.R.: Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580 (2012)
8. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)

9. Jin, W., et al.: Binary function clustering using semantic hashes. In: 2012 11th International Conference on Machine Learning and Applications (ICMLA), vol. 1, pp. 386–391. IEEE (2012)
10. Liebler, L., Baier, H.: Approxis: a fast, robust, lightweight and approximate disassembler considered in the field of memory forensics. In: Matoušek, P., Schmiedecker, M. (eds.) ICDF2C 2017. LNICST, vol. 216, pp. 158–172. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73697-6_12
11. Ligh, M.H., Case, A., Levy, J., Walters, A.: The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. Wiley, US (2014)
12. Lipton, Z.C., Berkowitz, J., Elkan, C.: A critical review of recurrent neural networks for sequence learning. arXiv preprint arXiv:1506.00019 (2015)
13. Potchik, B.: Architecture agnostic function detection in binaries. https://binary.ninja/2017/11/06/architecture-agnostic-function-detection-in-binaries.html
14. Shin, E.C.R., Song, D., Moazzezi, R.: Recognizing functions in binaries with neural networks. In: USENIX Security Symposium, pp. 611–626 (2015)